High-Speed Components And A Cache Memory Lower Access Times

In systems where total processing throughput is paramount, higher speed components are being employed and caching devices improve system performance significantly.

by **James Reinhart** and **Clara Serrano,** Motorola Inc.

As the sophistication of VLSI devices increases, the architectural complexity of microprocessor-based systems is rapidly approaching a level previously found only in large-scale computing systems. Specifically, advanced devices such as the MC68010 virtual memory processor and the MC68451 memory management unit provide for the implementation of high performance, multiuser, multitasking computer systems. A caching mechanism that provides high speed storage for often used information is one method of improving performance significantly.

INCREASING THROUGHPUT

The MC68010 processor is available with a maximum rated operating frequency of 12.5 MHz. However, increasing system throughput only partially depends on the CPU speed. Access time to external memory is the primary factor binding performance of high-speed microprocessors, and it's of little benefit to increase the CPU clock rate without a corresponding decrease in the average memory cycle time.

The M68000 family typically utilizes 90–95% of the external bus bandwidth (6.25 Mbytes/sec at 12.5 MHz) due to the highly efficient, pipelined internal architecture, and it can easily be demonstrated that the CPU performance is essentially bus-bound.

The average processing performance of the MC68010 is a function of the memory access time (Fig 1). The breaks in the performance curves occur at access times beyond which an additional wait state is introduced into the CPU bus cycle. The 12.5 MHz MC68010 has a minimum bus cycle time of 320 nsec (four 80 nsec clock periods). This is superior to the 10 MHz CPU (400 nsec minimum bus-cycle time) only if the average memory access times for the 12.5 MHz system are reduced to allow the faster CPU to run with an equivalent or lower number of wait states.

To perform a read cycle with no wait states, the 12.5 MHz MC68010 requires data valid at the processor inputs a maximum of 135 nsec after the assertion of the CPU address strobe ($\overline{AS}$). A 10 MHz device requires a 185 nsec access time for similar bus characteristics.

The inclusion of a VLSI memory-management device such as the MC68451 into an MC68010-based system provides hardware support for complex operating systems. This support may include the control of access in virtual paged or segmented environments while simplifying many tasks of the operating system.

The incorporation of a hardware memory management unit (MMU) complicates the design of a high performance system since all accesses to physical memory must now be translated through the MMU. The fastest MMUs available have a 120 nsec logical-to-physical translation time clearly indicating that a no-wait-state access to a large physical memory array is quite difficult, if not unrealistic, with a 12.5 MHz CPU.

To make use of higher speed CPUs, it's necessary to adjust the over-

Reprinted by permission of the publisher, from the Winter 1984 edition of COMPUTER TECHNOLOGY REVIEW (R)

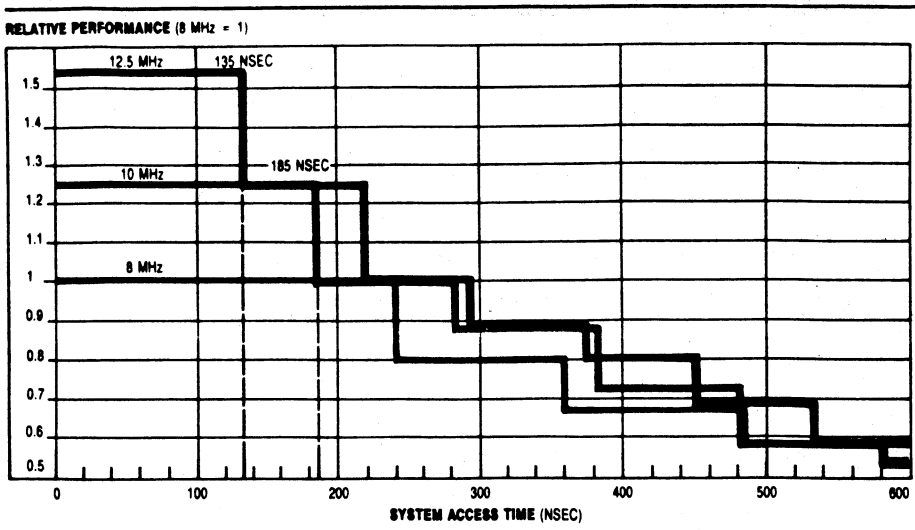

Fig 1 The performance of the MC68010 CPU compared to memory speed indicates that the processor is essentially bus-bound.

all system speed to support the increased processing capabilities. This may be achieved in one of two ways. First, the response time of the entire system can be reduced—including, at a minimum, reducing the access time to the main memory. This method is often preferred because of its conceptual simplicity. Faster memory devices are designed in, possibly making use of advanced features to reduce average access time.

Page-mode dynamic RAMs allow for reduced access times for successive memory cycles that access the same RAM row address. Also, nibble-mode RAMs provide serial access to 4 bits (a nibble) of data much faster than possible with standard memory devices. Newer technology control and buffer logic is utilized to reduce propagation delays, and faster MMUs can be used to lower translation delays. Translation look-aside buffers can also be employed to reduce average translation times further.

Decreasing the overall system response time is simple in concept and directly attacks the problem of increasing CPU throughput. However, this approach is limited by the cost of building an entire main store (1 Mbyte or more) with a sufficiently low access time to keep the CPU supplied with information at its maximum transfer rate.

The second approach for improving performance is to include a cache or buffer memory large enough and powerful enough to cut the average system access time to an acceptable level. In addition to reducing average system access time, a cache memory frees the main memory array for operations by bus masters other than the CPU (DMA controllers, for example) when the CPU bus cycles are accessing the cache. Cache memories have been utilized for many years in larger systems but, until recently, have not been widely used in microprocessor-based systems.

CACHE MEMORIES

A cache memory is a high-speed memory device that is placed between the CPU and the main memory (Fig 2). A cache memory can be located on either the logical bus, between the CPU and the MMU, or on the physical bus between the MMU and the main memory. The relative merits of each are discussed below.

The objective in implementation of a cache is to store information as it's first accessed by the processor and to provide this data very quickly if it's referenced again. Since almost all useful programs exhibit looping behavior to some extent, a properly designed cache captures all or some portion of these loops and provides the instruction and data much faster than if it had to be reread from the main store each time it was referenced.

The storage area of a cache is divided into two distinct regions (Fig 3). The data array stores the data values (operands, instructions, or both) associated with CPU bus cycles. Paired with each entry in the data array is a corresponding value in the tag array that stores, at a minimum, the full address (logical or physical) of the data value and a valid bit that indicates whether the entry contains valid information.

A range of the lower address bits can be used as an index into the tag and data arrays, and subsequently need not be included in the tag field. More versatile cache implementations also store status information such as the CPU function codes that indicate address space and privilege level, and a task ID field that identifies the task that caused the cache entry to be made.

Each time the processor initiates a bus cycle to transfer an operand, the tag-compare circuitry in the cache compares the access address and pertinent status information with that contained in the tag array. If the compare is successful, the cache is said to hit. Then the cache supplies the data value from its data array, and the access is completed. If the compare fails, then the bus cycle to main memory is allowed to continue and the corresponding cache entry is updated with the tag and data resulting from the completed bus cycle.

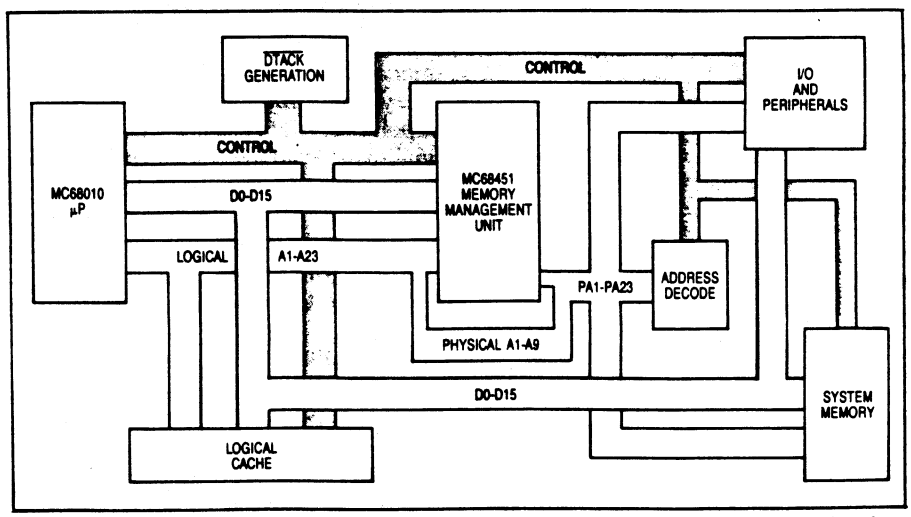

Fig 2 CPU performance can be improved in a high speed MC68010/MC68451 when a logical cache is between the processor and memory.

ASSOCIATIVITY

An address is mapped into a cache entry by a method referred to as cache associativity. If a given address can be mapped into only one location within the cache, the cache is considered direct-mapped (single-set associative). This implies that low-order address bits are used as indices to identify a unique location in the tag array. This location contains all higher order address bits and any other required information.

In a direct-mapped memory, all locations in the address space that have the same low-order address bits map into the same cache entry. Since the index field of the access address points to a unique cache location, a direct-mapped cache needs only a single comparator to determine whether the tag location contains valid information.

The counterpart of the direct-mapped cache is the fully associative or content-addressable cache which allows any address to be mapped into any location within the array. The entire access address is stored in the tag field (minus any bits used to select blocks within a data location) and when a bus cycle is initiated, the CPU outputs are checked simultaneously against all entries in the tag array. A content-addressable memory requires a comparator for each entry in the cache in order to perform the parallel compare operation and is necessarily limited in size by cost considerations.

A compromise between direct-mapped and content-addressable caches is the set-associative cache in which a given address can be mapped into two or more locations. The set size indicates the number of possible locations to which an address may be mapped. For instance, with a 4-set associative cache, an address may be mapped into one of four locations. The set size also dictates the number of simultaneous compare operations that must occur and, accordingly, the number of comparator elements that must be included.

Set-associative caches can help minimize excessive replacement of cache entries (known as thrashing) caused by programs that tend to access different operands frequently—operands whose addresses correspond to the same location in a direct-mapped cache. The cache works by allowing a fixed, lower-order address to map into more than one location.

In addition to multiple compare elements, semi- and fully associative caches require implementation of a replacement algorithm in order to determine which of several equivalent entries to update on a cache miss. Replacement algorithms vary in complexity and efficiency from the most efficient (and costly) 'least recently used' method which attempts to create a pseudo working-set list of data operands, to the random-replacement method which arbitrarily selects an entry to be replaced.

In-house studies conducted at Motorola indicate that for small to medium-sized caches (256 bytes to 4 Kbytes) a small degree of associativity (2-, 4-, or 8-set) provides for only a 1–2% performance increase over a similarly sized, direct-mapped cache. Therefore, for the implementation discussed here, it was decided not to pursue construction of an associative cache because of the marginal return in doing so.

BLOCK SIZE

The block size of a cache refers to the amount of information, usually in number of bytes, stored in each location of the data array. In most implementations, the block size also equals the amount of data loaded into the cache during a replacement operation. The cache under consideration here was implemented with block and replacement set sizes of 2 bytes, which corresponds to the data bus width of the MC68010 processor.

Although increasing the block size without increasing the replacement set size produces no discernible performance enhancement, increasing both sizes equally provides an improvement in cache hit ratios. While the main memory width for this study was fixed at 16 bits, simulations show that utilizing a 32-bit data path to main memory and updating the cache with 4 bytes on each miss provides an average 10–15% increase in hit rates. For many high performance systems, the additional cost of implementing a 32-bit main memory and the extra control circuitry required to implement this type of cache may be justified by the increase in overall system performance.

WRITE-REQUEST HANDLING

The inclusion of a cache creates a 2-level memory hierarchy (3-level, if disk storage is also considered) that must be in agreement concerning the correct value of a memory location. Decoupling of the CPU from the main memory enables different bus masters to access different parts of the memory hierarchy simultaneously and, potentially, to access the same operands. If a data value in the cache corresponding to a physical memory location differs from a data value in the main store, the system must be able to determine which of the copies contains the most recent data.

The determination of a write strategy dictates what special requirements, if any, are necessary to minimize stale data problems. The cache control can either mark the entries that have been modified in the cache and write them into main memory when a cache miss occurs (write-back), or it can write modified data to both the cache and the main memory on every write cycle (write-through).

The write-back scheme requires maintenance of additional tag bits to identify altered entries in the cache, and additional control hardware to perform the memory update. The preferred method for handling writes is to use write-through and force each write access to update both copies of the operand.

Often a controversial issue, the determination of cache placement should be divided into an evaluation of the relative trade-offs between logical and physical caches. The primary goal of the cache system designer is to minimize the average memory access time while minimizing operating system overhead for cache maintenance. The main operating system concern in manipulating the cache is to resolve problems involving stale data.

PHYSICAL ADDRESS CACHE

The physical address cache benefits primarily from the use of physical (real) addresses to map information into the cache. This procedure allows the cache controller to monitor all activity that affects a physical memory location because a unique address is generated to identify this location. A virtually unlimited number of logical addresses could be generated, all of

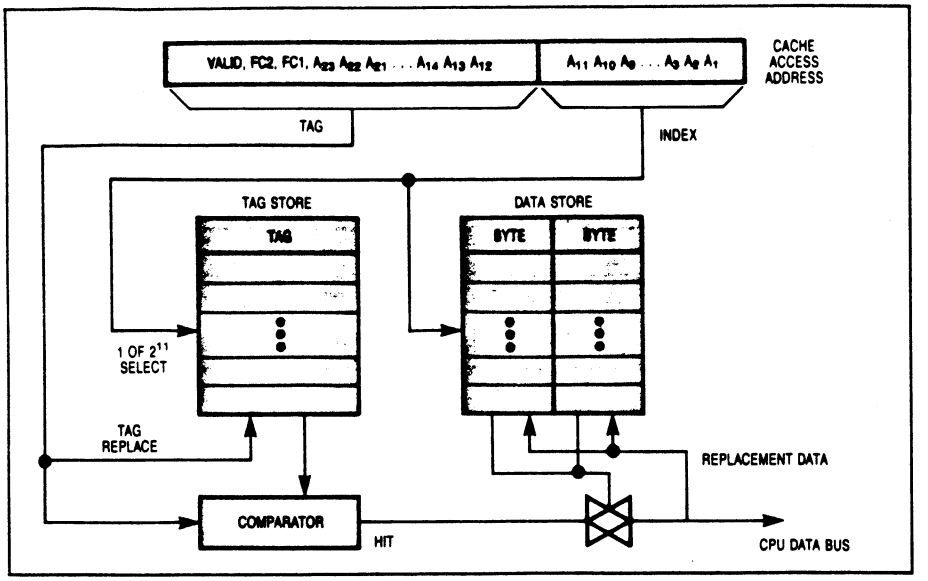

Fig 3 In this 4 Kbyte direct-mapped cache (block size of 1), the data array stores the data values associated with CPU bus cycles. A corresponding value in the tag array is paired with each entry in the data array.

which reference the same physical memory location.

The ability to have hardware (often called a DMA monitor) observe all accesses that affect physical locations without regard to the source of the access (translated CPU access, DMA activity, etc.) resolves all stale data problems without requiring operating system intervention. However, implementation of a DMA monitor requires the monitor to have access priority to the cache when CPU and DMA accesses occur simultaneously.

High DMA activity can leave the CPU waiting indefinitely for cache access, defeating the secondary advantage of cache memory—that of keeping the CPU supplied with data and the main memory free for operations by other bus masters (such as DMA controllers). Clearly, careful consideration must be given to estimations of DMA activity rates in order to determine the performance degradation that might occur.

The primary disadvantage with the implementation of a physical cache is that all CPU accesses must suffer the MMU translation time. A 12.5 MHz MC68010 requires assertion of the data transfer acknowledge signal ($\overline{DTACK}$) 40 nsec after the address strobe ($\overline{AS}$) is asserted in order to perform a no-wait-state bus cycle. Based on a 120-nsec, worst-case translation time for the MMU, it's clearly not feasible for a physical cache to compare the translated address and determine (in time to signal the completion of a no-wait-state bus cycle) whether a hit occurred.

LOGICAL CACHE

The most important benefit of locating a cache memory on the logical bus is speed of access. A cache entry is indicated at the end of a bus cycle to physical memory, and the entry loaded into the tag array corresponds to the logical address generated by the CPU. During subsequent bus cycles, the cache tag-compare logic compares the logical address with the tag array contents. Without waiting for the result of this comparison, the MMU initiates a translation, reducing access time if a cache miss occurs.

Similarly, the compare logic doesn't wait for translation by the MMU. If a cache hit occurs, the MMU cycle is terminated and the cache supplies data to the CPU. A sufficiently fast cache controller allows the CPU to operate with no wait states when hitting in the logical cache.

Complications with logical cache designs center around the stale data problem. Since the logical cache doesn't monitor physical accesses, it's possible that another physical bus master could update the data value in main memory and cause the cache entry to become stale.

A physical DMA bus monitor for a logical cache can be implemented if you store, in the tag array, both the logical and physical addresses that correspond to cache entries. This approach allows for asynchronous operation of the logical and physical buses but requires additional tag storage in order to store both the logical and physical addresses.

To overcome the stale data problem in a logical cache, a simple operating system convention can be followed concerning a task's relationship with I/O operations initiated for that task. When DMA activities have been allowed into a task's address map, the cache should be flushed prior to allowing that task to access the new information. Flushing the cache ensures that entries loaded into the cache contain the most recent data values.

Similarly, task-related information in the cache should be flushed by the task dispatch handler on execution of a context switch. These responsibilities are assumed by the operating system and become a part of its normal task management duties.

SYSTEM IMPLEMENTATION

The system described here incorporates a 12.5-MHz MC68010 microprocessor translating through a 10-MHz MC68451 memory management unit. Use of a higher speed CPU required some manipulation of the bus control strobes to ensure correct synchronization. A direct-mapped logical data cache was designed with the primary goal being to average less than one wait state for all bus cycles. Accesses to main memory were designed to incur no more than two wait states—indicating that a no-wait-state cache must provide a 50% or greater hit rate to realize the system average of less than one wait state.

The 1-Mbyte main memory was constructed by use of 150-nsec MCM6665 64K dynamic RAMs. To perform a two-wait-state physical access, it was necessary to minimize the propagation delays in the path to main memory, and FAST (Fairchild Advanced Schottky TTL) logic was used extensively for speed-critical buffers and line drivers.

The logical cache was built with eight TMS2150-45s for the tag store and compare, and two MCM2016-75 2K × 48 static RAMs for the data store. The requirement that the cache support no wait-state bus operations for cache hits dictated that the cache be very closely synchronized to the CPU. Cache compare timing did not allow for the inclusion of buffering delays and, therefore, the maximum cache size was determined by the ca-

pacitive loading limits of the MC68010 buses.

A cache inhibit signal from the physical address decode is used to prevent loading accesses made to the I/O and peripheral address space into the cache. Other control information decoded from physical addresses includes cache enable and clear signals from control locations in the supervisor address space. The 4-Kbyte cache configuration required a total of 20 integrated circuits.

TABLE — SYSTEM PERFORMANCE FOR VARIOUS CACHE ORGANIZATIONS

CACHE SIZE	QUEENS				QUICKSORT			
	HIT RATIO (%)		TIME (SEC)		HIT RATIO (%)		TIME (SEC)	
NO CACHE	0		38.06		0		31.17	
256 BYTES DATA INSTRUCTION ONLY	41.8	32.0	32.64	33.87	41.2	33.9	26.90	27.65
512 BYTES DATA INSTRUCTION ONLY	52.9	43.9	31.34	32.48	46.3	34.7	26.36	27.56
1024 BYTE DATA INSTRUCTION ONLY	52.9	43.9	31.34	32.48	50.9	38.0	25.87	27.10
2048 BYTES DATA INSTRUCTION ONLY	61.6	43.9	30.23	32.48	56.9	43.6	25.28	26.64
4096 BYTES DATA INSTRUCTION ONLY	61.6	43.9	30.08	32.48	59.1	43.6	22.86	26.64

CACHE PERFORMANCE EVALUATION

Various benchmark programs were executed on the 68010/68451 logical cache system to evaluate the performance of memory-intensive and localized programs. The effect of cache size was studied as well as the performance of an instruction-only cache compared to a data cache (instructions and data are both brought into the cache).

Two of these benchmark programs are Queens and Quicksort. The Queens program takes an n × n chessboard (9 × 9 in this example) and finds the placement of the maximum number of queens such that they don't attack each other. Using exhaustive iteration, the program steps through all the possible combinations and stops when it finds all solutions.

Although it executes in relatively short loops, exhibiting significant locality of reference, Queens tends to generate several logical addresses that map to the same cache location (thrashing). This tendency, which causes frequent replacement of the entries associated with these addresses and a corresponding decrease in hit rates, illustrates one of the factors limiting the performance of relatively small direct-mapped caches.

The Quicksort program searches through 15,000 long-word (32-bit) integers in memory and sorts them into ascending order. It steps through the array with two pointers, A and B, until the value at B is smaller than the value at A. Next, it swaps the two entries and calls itself again with new bounds. The program continues recursively until all 15,000 entries have been sorted. Thus, Quicksort accesses a larger data area than that used by Queens. The Quicksort program itself is also larger than the Queens program, and the runtime characteristics show decreased locality of reference—as expected.

All benchmarks are written in Pascal and compiled into MC68010 object code to reproduce the operations of typical high-level-language environments more closely. The system environment is similar to what could be expected in a multiuser/multitasking system. Note that assembly language benchmarks could have been written to take further advantage of the cache and achieve hit ratios approaching 100%, but that isn't realistic for typical high-level-language programs.

Benchmark results were obtained for each cache configuration by counting the number of physical memory bus cycles occurring during execution of Queens and Quicksort with the cache turned on. This number was subtracted from the total number of bus cycles necessary to execute these programs (the total was obtained by turning off the cache) yielding the total number of memory cycles that hit in the cache.

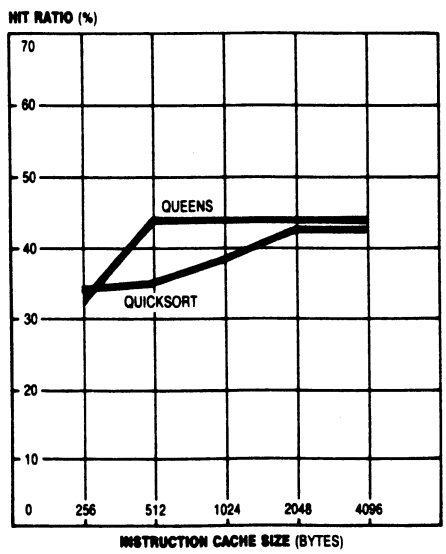

Fig 4 The Queens program is fully optimized at a cache size of 512 bytes, and no real performance improvement is obtained by increasing the instruction cache size beyond this limit.

Hit ratios were then calculated, along with execution times based on a 6-clock cycle of 480 nsec for physical memory accesses (two wait-state cycles at 80 nsec/cycle) and a 4-clock cycle of 320 nscc for memory accesses that hit in the cache (no wait-state cycle at 80 nsec/cycle).

The configuration for an instruction-only cache is similar to that shown in Fig 3 except that the MC68010 FCO signal (a logic level ONE during data accesses) was used to disable the cache for all data accesses. The cache size was varied from 256 bytes to 4096 bytes, yielding the results shown in Fig 4 and the Table.

Quicksort is optimized at a cache size of 2048 bytes, and Queens is optimized at 512 bytes. This finding is expected since Quicksort object code is larger than that of the compiled Queens program. Clearly, larger programs benefit from still larger instruction-only caches.

DATA CACHE

Analysis of the test results shows that as the cache size was increased from 256 to 512 bytes, the Queens program exhibited an 11% increase in hit ratio (Fig 5). Comparing this slope with the corresponding graph of Fig 4, it's evident that the 11% increase can be attributed to additional instruction fetches hitting in the cache. The second increase in hit ratio between cache sizes of 1024 and 2048 bytes (another 11%), however, can be attributed to a significant increase in the number of data accesses made from the cache at 2048 bytes compared to the number accessed from the cache at a size of 1024 bytes.

Alternately, the Quicksort program performance (hit ratio) tends to increase almost linearly with cache size in a data cache configuration. This was expected, since the program manipulates a very large array (15,000

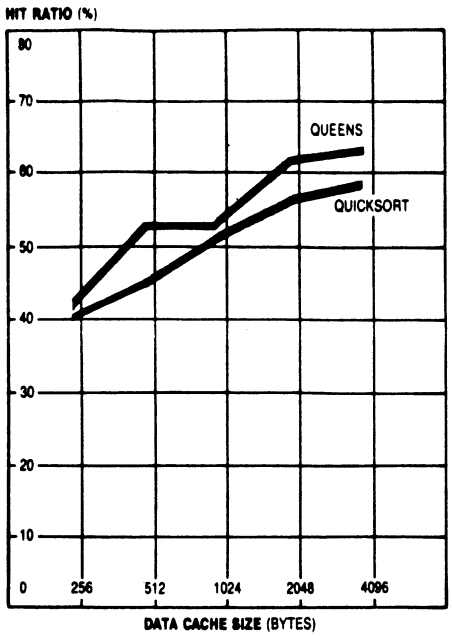

Fig 5 When the Queens program was run, the hit ratio for a data cache improved 11% when the cache size was increased from 1024 to 2048 bytes.

data entries). As expected, when cache size is increased, progressively more of the data accesses hit in the cache.

By comparing the relative hit ratios of Fig 4 with those of Fig 5, the trade-offs inherent in implementation of an instruction-only cache rather than a data cache can be evaluated. As shown in the figures, a data cache yields an average of 12% higher hit ratios than an instruction-only cache. Note also that the hit ratio of Queens, with an instruction-only cache of 4096 bytes (43.9%), is only 3% higher than the hit ratio with a data cache of 256 bytes (41.2%).

From these results, it's evident that the choice of an optimum cache size is highly application-dependent. For larger programs, larger instruction caches improve hit ratios. For programs that access a large range of addresses for data, larger data caches result in increased hit ratios.

A physical cache can alleviate stale data problems without requiring intervention from the operating system, but often at the expense of reduced performance. The logical cache offers lower access times than the physical cache but requires the operating system to be aware of the existence of the cache and flush entries when required.

The cache size and organization also affect the price/performance ratio. The marginal return for doubling the size of a direct-mapped cache decreases significantly above 4 Kbytes, indicating that this type of cache is approaching its optimal size. A larger cache would benefit from some degree of associativity.

The minimum cache structure that satisfies Motorola's performance goals is the 2-Kbyte data cache. The 4-Kbyte configuration was used to maintain high hit rates with larger data structures than evaluated with the test cases used. The direct-mapped architecture provided sufficient hit ratios to satisfy the design goals of averaging less than one wait state per bus cycle.

Future versions of this logical cache, with hit rate requirements of 80% and above, will require implementation of a larger (8–16 Kbyte) data storage array and a small degree of associativity (2- or 4-set). A 32-bit path to main memory will be implemented to further decrease the number of cache misses. ■

James Reinhart *is system applications engineer for Motorola Inc. and has a BSEE from Rice Univ.*

Clara Serrano *is an applications engineer for Motorola Inc. and obtained her BSEE in 1984 from the Univ. of Wisconsin.*